FPH: Efficient Non-commutativity Analysis of Feature-Based Systems

Marsha Chechik[1(✉)], Ioanna Stavropoulou[1], Cynthia Disenfeld[1], and Julia Rubin[2]

[1] University of Toronto, Toronto, Canada
{chechik,ioanna,disenfeld}@cs.toronto.edu
[2] University of British Columbia, Vancouver, Canada
mjulia@ece.ubc.ca

Abstract. *Feature-oriented software development (FOSD)* is a promising approach for developing a collection of similar software products from a shared set of software assets. A well-recognized issue in FOSD is the analysis of *feature interactions*: cases where the integration of multiple features would alter the behavior of one or several of them. Existing approaches to feature interaction detection require a fixed order in which the features are to be composed but do not provide guidance as to how to define this order or how to determine a relative order of a newly-developed feature w.r.t. existing ones. In this paper, we argue that classic feature non-commutativity analysis, i.e., determining when an order of composition of features affects properties of interest, can be used to complement feature interaction detection to help build orders between features and determine many interactions. To this end, we develop and evaluate Mr. Feature Potato Head (FPH) – a modular approach to non-commutativity analysis that does not rely on temporal properties and applies to systems expressed in Java. Our experiments running FPH on 29 examples show its efficiency and effectiveness.

1 Introduction

Feature-oriented software development (FOSD) [3] is a promising approach for developing a collection of similar software products from a shared set of software assets. In this approach, each feature encapsulates a certain unit of functionality of a product; features are developed and tested independently and then integrated with each other; developed features are then combined in a prescribed manner to produce the desired set of products. A well-recognized issue in FOSD is that it is prone to creating *feature interactions* [2,13,22,28]: cases where integrating multiple features alters the behavior of one or several of them. Not all interactions are desirable. E.g., the Night Shift feature of the recent iPhone did not allow the Battery Saver to be enabled (and the interaction was not fixed for over 2 months, potentially affecting millions of iPhone users). More critically, in 2010, Toyota had to recall hundreds of thousands of Prius cars due to an

A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 319–336, 2018.
https://doi.org/10.1007/978-3-319-89363-1_18

interaction between the regenerative braking system and the hydraulic braking system that caused 62 crashes and 12 injuries.

Existing approaches for identifying feature interactions either require an explicit order in which the features are to be composed [6,8,18,19,26] or assume presence of a "150%" representation which uses an implicit feature order [12,15]. Yet they do not provide guidance on how to define this order, or how to determine a relative order of a newly-developed feature w.r.t. existing ones.

A classical approach of feature non-commutativity detection, defined by Plath and Ryan [25], can be used to help build a composition order. The authors defined non-commutativity as "the presence of a property, the value of which is different depending on the order of the composition of the features" and proposed a model-checking approach allowing to check available properties on different composition orders. E.g., consider the Elevator System [14,25] consisting of five features: *Empty* – to clear the cabin buttons when the elevator is empty; *ExecutiveFloor* – to override the value of the variable `stop` to give priority to the executive floor (not stopping in the middle); *TwoThirdsFull* – to override the value of `stop` not allowing people to get into the elevator when it is two-thirds full; *Overloaded* – to disallow closing of the elevator doors while it is overloaded; and *Weight* – to allow the elevator to calculate the weight of the people inside the cabin. Features *TwoThirdsFull* and *ExecutiveFloor* are not commutative (e.g., a property "the elevator does not stop at other floors when there is a call from the executive floor" changes value under different composition orders), whereby *Empty* and *Weight* are. Thus, an order between *Empty* and *Weight* is not required, whereas the user needs to determine which of *TwoThirdsFull* or *ExecutiveFloor* should get priority. Thus, *feature non-commutativity guarantees a feature interaction, whereas feature commutativity means that order of composition does not matter. Both of these outcomes can effectively complement other feature interaction approaches.*

In this paper, we aim to make commutativity analysis practical and applicable to a broad range of modern feature-based systems, so that it can be used as "the first line of defense" before running other feature interaction detections. There are three main issues we need to tackle. First of all, to prove that features commute requires checking their composition against all properties, and capturing the complete behavior of features in the form of formal specifications is an infeasible task. Thus, we aim to make our approach *property-independent*. Second, we need to make commutativity analysis *scalable* and avoid rechecking the entire system every time a single feature is modified or a new one is added. Finally, we need to support analysis of systems expressed in modern programming languages such as Java.

In [25], features execute "atomically" in a state-machine representation of the system, i.e., they make all state changes in one step. However, when systems are represented in conventional programming languages like Java, feature execution may take several steps; furthermore, such features are composed *sequentially*, using *superimposition* [5]. Examining properties defined by researchers studying such systems [6], we note that they do not refer to intermediate states within

the feature execution, but only to states before or after running the feature, effectively treating features as atomic. In this paper, we use this notion of atomicity to formalize commutativity. The foundation of our technique is the separation between feature behavior and feature composition and efficiently checking whether different feature compositions orders leave the system in the same internal state. Otherwise, a property distinguishing between the orders can be found, and thus they do not commute. We call the technique and the accompanying tool Mr. Feature Potato Head (*FPH*), named after the kids' toy which can be composed from interchangeable parts.

In this paper, we show that FPH can perform commutativity analysis in an efficient and precise manner. It performs a modular checking of *pairs of features* [17], which makes the analysis very scalable: when a feature is modified, the analysis can focus only on the interactions related to that feature, without needing to consider the entire family. That is, once the initial analysis is completed, a partial order between the features of the given system can be created and used for detecting other types of interactions. Any feature added in the future will be checked against all other features for non-commutativity-related interactions to define its order among the rest of the features, but the existing order would not be affected. In this paper, we only focus on the non-commutativity analysis and consider interaction *resolution* as being out of scope.

Contributions. This paper makes the following contributions: (1) It defines commutativity for features expressed in imperative programming languages and composed via superimposition. (2) It proposes a novel modular representation for features that distinguishes between feature composition and behavior. (3) It defines and implements a modular specification-free feature commutativity analysis that focuses on pairs of features rather than on complete products or product families. (4) It instantiates this analysis on features expressed in Java. (5) It shows that the implemented analysis is effective for detecting instances of non-commutativity as well as proving their absence. (6) It evaluates the efficiency and scalability of the approach.

The rest of the paper is organized as follows. We provide the necessary background, fix the notation and define the notion of commutativity in Sect. 2. In Sect. 3, we describe our iterative tool-supported methodology for detecting feature non-commutativity for systems expressed in Java. We evaluate the effectiveness and scalability of our approach in Sect. 4, compare our approach to related work in Sect. 5 and conclude in Sect. 6[1].

2 Preliminaries

In this section, we present the basic concepts and definitions and define the notion of commutativity used throughout this paper.

[1] The complete replication package including the tool binary, case studies used in our experiments and proofs of selected theorems is available at https://github.com/FeaturePotatoHead/FPH.

```
1  package ElevatorSystem;
2  public class Elevator {
3    int executiveFloor = 4;
4    public boolean isExecutiveFloor(int floorID) {...}
5    public boolean isExecutiveFloorCalling () {...}
6    private boolean stopRequestedAtCurrentFloor() {...}
7    private boolean stopRequestedInDirection (Direction
         dir, boolean respectFloorCalls, boolean
         respectInLiftCalls ) {
8    if ( isExecutiveFloorCalling ()) { ... }
9    else return original (dir, respectFloorCalls,
         respectInLiftCalls );
10 }}
```

Fig. 1. Java code snippet of the feature *ExecutiveFloor*.

Feature-Oriented Software Development (FOSD). In FOSD, *products* are specified by a set of features (*configuration*). A *base system* has no features. While defining the notion of a feature is an active research topic [11], in this paper we assume that a feature is *"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision and to offer a configuration option"* [5].

Superimposition. *Superimposition* is a feature composition technique that composes software features by merging their corresponding substructures. Based on superimposition, Apel et al. [5] propose a composition technique where different components are represented using a uniform and language independent-structure called a *feature structure tree (FST)*. An *FST* is a tree $T = \langle$(Terminal Node) | (Non Terminal Node) (Tree T)+$\rangle$, where + denotes "one or more". A *Non Terminal Node* is a tuple $\langle name, type \rangle$ which represents a non-leaf element of T with the respective name and type. A *Terminal Node* is a tuple $\langle name, type, body \rangle$ which represents a leaf element of T. In addition to *name* and *type*, each *Terminal Node* has *body* that encapsulates the content of the element, i.e., the corresponding method implementation or field initializer. A *feature* is a tuple $f = \langle name, T \rangle$, where *name* is a string representing f's name and T is an FST abstractly representing f.

Each feature describes the modifications that need to be made to the base system, also represented by an FST, to enable the behavior of the feature. While FSTs are generally language-independent, in this paper we focus on features defined in a Java-based language. For example, consider the Java code snippet in Fig. 1, which shows the *ExecutiveFloor*. This feature makes one of the floors "an executive one". If there is a call to or from this floor, it gets priority over any other call. This feature is written in Java using a special keyword `original` [5] (line 9). Under this composition, a call from the new method to every existing method with the same name is added, in order to preserve the original behavior. Without `original`, new methods replace existing ones.

The feature *ExecutiveFloor* in Fig. 1 is represented by the tuple $\langle executive, T \rangle$, where T is the FST in Fig. 2. `ElevatorSystem` is a Non Terminal Node that represents the `ElevatorSystem` package with the tuple $\langle$`ElevatorSystem`, `package`$\rangle$, and `stopRequestedInDirection` is a Terminal Node represented by $\langle$`stopRequestedInDirection`, `method`, *body*$\rangle$, where *body* is the content of the `stopRequested-InDirection` method in Fig. 1 (lines 8–9). Another Non Terminal Node is `Elevator`, whereas `executiveFloor`, `isExecutiveFloor`, `isExecutiveFloorCalling` and `stopRequestedAtCurrentFloor` are Terminal.

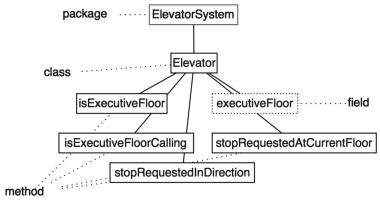


Fig. 2. FST representation for the feature *ExecutiveFloor*.

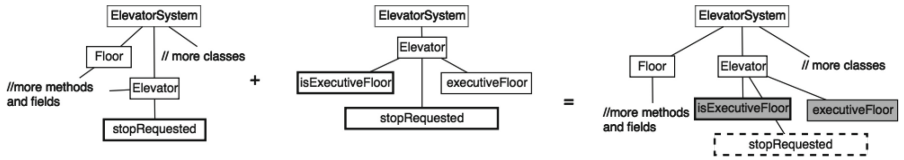


Fig. 3. Simplified composition of *ExecutiveFloor* and the base elevator system.

For Java-specified features, Terminal Nodes represent methods, fields, `import` statements, modifier lists, as well as `extends`, `implements` and `throws` clauses whereas directories, files, packages and classes are represented by Non Terminals.

Superimposition Process. Given two FSTs, starting from the root and proceeding recursively to create a new FST, two nodes are composed when they share the same name and type and when their parent nodes have been composed. For Terminal Nodes which additionally have a body, if a Node A is composed with a Node B, the body of A is replaced by that of B unless the keyword `original` is present in the body of B. In this case, the body of A is replaced by that of B and the keyword is replaced by A's body. Since the `original` keyword is not used for fields, the body of the initial field is always replaced by that of the new one.

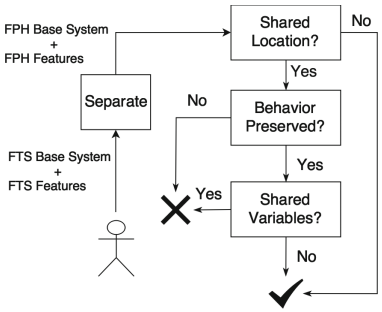
Figure 3 gives an example of a composition of a simplified *ExecutiveFloor* feature with the elevator base system. Terminal Nodes that have been overridden by the feature are with dashed outline and new fields and methods added by the feature are shown as shaded nodes. For example, the method `stopRequested`, which is part of the base system, is overridden by the feature, whereas the field `executiveFloor`, which is only part of the feature, is added to the base system.

Commutativity. We define commutativity w.r.t. properties observable before or after features finish their execution (as those in [6]). A *state* of the system after superimposing a feature is the valuation of each variable (or array, object, field, etc. [24]) of the base system and each variable (or array, etc.) introduced by the feature. We also add a new variable *inBase* which is *true* iff this state is not within a method overridden by any feature. In the rest of the paper, we refer to states where *inBase* is *true* as *inBase* states. A *transition* of the system is an execution of a statement, including method calls and return statements [24].

Then we say that two features *commute* if they preserve valuation of properties of the form $G(inBase \implies \phi)$, where ϕ is a propositional formula defined over any system state variables. That is, they do not commute if there is at least one state of the base system which changes depending on the order in which the features are composed. For example, the property "the elevator does not stop at other floors when there is a call from the executive floor", used in Sect. 1 to identify non-commutativity between features *TwoThirdsFull* and *ExecutiveFloor*, is $G(inBase \implies \neg(isExecutiveFloorCalling \wedge stopped \wedge floor \neq executiveFloor))$.

3 Methodology

Our goal is to provide a scalable technique for determining whether features commute by establishing whether the two different composition orders leave the system in the same internal state. The workflow of FPH is shown below. The first step of FPH is to transform each feature from an FST into an FPH representation consisting of a set of fragments. The base is transformed in the same way as the individual features. Each fragment is further split into feature behavior and feature composition – see Sect. 3.1. Afterwards, we check for non-compositionality. If there do not exist feature fragments that have *shared location* of composition, i.e., whose feature composition components are the same, then the features commute. Otherwise, check the pairs of feature fragments for *behavior preservation*, i.e., when the two features are composed in the same location, the previous behavior is still present and can be executed. If this check succeeds, we perform the *shared variables* check – see Sect. 3.2.



3.1 Separating Feature Behavior and Composition

We now formally define the FPH representation of features that separates the behavior of features and location of their composition and provide transformation operators between the FPH and the FST representations.

Definition 1. *An FPH feature is a tuple* ⟨name, fragments⟩*, where* name *is the feature name and* fragments *is the list of feature fragments that comprise the feature. Let a feature f be given. A* Feature Fragment fg *is a tuple* ⟨fb, fc⟩*, where* fb *is a feature behavior defined in Definition 2 and* fc *is a feature composition defined in Definition 3.*

Definition 2. Feature Behavior fb *of a feature fragment* fg *is a tuple* ⟨name, type,body, bp, vars⟩*, where* name*,* type *and* body *represent the name, type and content, respectively, of the element represented by* fg. bp *is a boolean value which is set to* true *if the feature preserves the original behavior, i.e., when the keyword* `original` *is present in the body and not within a conditional statement.* vars *is a list of variable names read or written within* fg.

Definition 3. Feature composition fc *of a feature fragment* fg *is represented by* ⟨location⟩ *which is the path leading to the terminal node represented by* fg.

The *Separate* operator (see Fig. 4a) transforms features from the FST to the FPH representation by creating a new fragment for each Terminal Node in the given FST. For the behavior component of the fragment, its *name*, *type* and *body* attributes come from the respective counterparts of the FST Terminal Node. The *bp* field is *true* if every path within *body* contains the keyword `original`; otherwise, it is *false*. For the composition component, the *location* field gets its value from the unique path to the Terminal Node from the root of the FST. *vars* are the parameters of the method and the fields that are used within it.

E.g., consider creating the FPH representation for *ExecutiveFloor* feature in Fig. 2. Since there are five Terminal Nodes, five fragments will be created to represent each node. In the fragment created for the `stopRequestedInDirection` node, the information in *fb* about *name*, *node* and *type* is derived from the information stored in the node, $fb = \langle$`stopRequestedInDirection`, method, [body]$\rangle$, where body consists of lines 8–9 of Fig. 1. *bp* is *false* since the keyword `original` is within an if statement and *vars* consists only of the method parameters since the method does not use any global fields. After separating, the feature composition is $fc =$ ElevatorSystem.Elevator.stopRequested-InDirection.

To transform features from FPH back to FST, we define the *Join* operator. It takes as input a list of feature fragments and returns an FST (see Fig. 4b).

```
Input: FST Representation of F
Output: Fragments of F {f = (fb, fc)}
1  begin
2      fragment list ← []
3      forall Non Terminal Nodes n ∈ FST do
4          f = (fb, fc) ← new Feature Fragment
5          fc.location ← n.name
6          fb.type ← n.type
7          fb.body ← n.body
8          fb.vars ← n.get-Variables()
9          if fb.body contains original in every path then
               fb.bp ← true
10         else fb.bp ← false
11         add f to fragment list
12     return fragment list
```

(a)

```
Input: Set of fragments of F {(fb, fc)}
Output: FST representation of F
1  begin
2      forall (fb, fc) ∈ F do
3          t ← new Terminal Node
4          t.name ← fb.name
5          t.type ← fb.type
6          t.body ← fb.body
7          forall node ∈ F do
8              if node ∉ FST then add node to FST
9              else continue
10     return FST
```

(b)

```
Input: Fragments of F1 and F2 {fj = (fbj, fcj)} with j ∈ {1, 2}
Output: Yes if F1 and F2 commute, No otherwise
1  begin
2      forall (fb1, fc1) ∈ F1, (fb2, fc2) ∈ F2 do
3          if (fc1 = fc2) then
4              if (fb1.bp = false ∨ fb2.bp = false) then
5                  return No
6              if (fb1.vars ∩ fb2.vars) ≠ ∅ then
7                  return No
8      return Yes
```

(c)

Fig. 4. Algorithms *Separate*, *Join* and *CheckCommutativity*.

It creates a new Terminal Node to be added to the FST for each feature fragment in the given feature. The *name*, *type* and *body* attributes of the node are filled using the corresponding fields in the feature behavior component of the fragment. Then, starting from the root node, for every node in the *location* path of the feature composition component, if the node does not exist in the FST, it is added; otherwise, the next node of the path is examined. The information about *bp* and *vars* is already contained in the body of the Terminal Node and is no longer considered as a separate field. E.g., joining the *ExecutiveFloor* feature that we previously separated yields the FST in Fig. 2, as expected.

Theorem 1. *Let n be the number of features in a system. For every feature F which can be represented as (fb, fc),* Join *and* Separate *are inverses of each other, i.e.,* Join*(*Separate*$(F)) = F$ and* Separate*(*Join*$(fb,fc)) = (fb,fc)$.*

3.2 Compositional Analysis of Non-commutativity

We now formally present the algorithm *check commutativity*, a sequence of increasingly more precise, and more expensive, static checks to perform non-commutativity analysis. These are called *shared location*, *behavior preservation* and *shared variables* – see Fig. 4c. Additionally, we prove soundness and correctness of the FPH methodology, i.e., that our checks guarantee feature commutativity as defined in Sect. 2.

Check Shared Location. The first check examines whether F_1 and F_2 have any fragments that can be composed in the same location (line 3). Clearly, when F_1 and F_2 are applied in different places, e.g., they change different methods, *inBase* states are the same independently of their order of composition, and thus the features commute. Otherwise, more precise checks are required. E.g., *ExecutiveFloor* (see Fig. 2) and *Empty* (see Fig. 5a) do not share methods or fields and thus can be applied in either order.

Theorem 2. *If features F_1 and F_2 are not activated in the same location, any* inBase *state resulting from first composing F_1 followed by F_2 (denoted $F_1; F_2$) is the same as for $F_2; F_1$.*

Check Behavior Preservation. Suppose one pair of feature fragments of F_1 and F_2, say, f_1 and f_2, can be composed in the same location. Then we examine whether the original behavior is preserved or overridden (indicated by the fb

Fig. 5. Two features of the elevator system.

field of each fragment). If *bp* of f_1 and f_2 is *true*, an additional check for shared variables is applied. Otherwise, i.e., when *bp* of either f_1 or f_2 is *false*, we report an interaction. Clearly, this check can introduce false positives because we do not look at the content of the methods but merely at the presence of the `original` keyword. E.g., two methods may happen to perform the exact same operation and yet not include the `original` keyword. In this case, we would falsely detect an interaction[2].

Check Shared Variables. If F_1 and F_2 are activated at different places and both preserve the original behavior, commutativity of their composition depends on whether they have shared variables that can be both read and written. This check aims to detect that. E.g., both features *Empty* (see Fig. 5a) and *Weight* (see Fig. 5b) modify the `leaveElevator` method and preserve the original behavior. Since no variables between them are shared, the order of composition does not affect the execution of the resulting system.

Extracting shared variable information requires not only identifying which variable is part of each feature behavior, but also running points-to analysis since aliasing is very common in Java. Moreover, a shared variable might not appear in the body of the affected method but instead in the body of a method called by it. Yet existing frameworks for implementing interprocedural points-to analyses [21] may not correctly identify all variables read and written within a method. Moreover, even if two features do write to the same location, this may not manifest a feature interaction. E.g., they may write the same value. For these reasons, our shared variables check may introduce false positives and false negatives. We evaluate its precision in Sect. 4.

Theorem 3. *Let features F_1 and F_2 activated at the same place and preserving the behavior of the base be given. If the variables read and written by each feature are correctly identified and independent of each other ($F_1.vars \cap F_2.vars = \emptyset$), then any* inBase *state resulting from composing $F_1; F_2$ is the same as that of composing $F_2; F_1$.*

When two features merely read the same variable, it does not present an interaction problem. We handle this case in our implementation (see Sect. 4).

Theorem 4 (Soundness). *Given features F_1 and F_2, if variables read and written by them are correctly identified, Algorithm in Fig. 4c is sound: when it outputs Success, F_1 and F_2 commute.*

Complexity. Let $|F|$ be the number of features in the system and let M be the largest number of fragments that each feature can have. For a pair of feature fragments, checking shared location and checking behavior preservation are both done in constant time, so the overall complexity of these steps is $O((|F| \times M)^2)$. In the worst case, all features affect the same set of methods and thus the shared variables check should be run on all of them. Yet, all fragments in a feature are non-overlapping, and thus the number of these checks is at most $|F|^2 \times M$.

[2] But this does not happen often – see Sect. 4.

The time to perform a shared variable check, which we denote by SV, can vary depending on an implementation and can be as expensive as PSPACE-hard. Thus, the overall complexity of non-commutativity detection is $O((|F| \times M)^2 + SV \times |F|^2 \times M)$.

4 Evaluation

In this section, we present an experimental evaluation of FPH, aiming to answer the following research questions: **(RQ1)** How effective is FPH in performing non-commutativity analysis of feature-based systems? **(RQ2)** How accurate is FPH's non-commutativity analysis? **(RQ3)** How efficient is FPH compared to state-of-the-art tools for performing non-commutativity analysis? **(RQ4)** How well does FPH scale as the number of fragments increases?

Tool Support. We have implemented our methodology (Sect. 3) as follows. The *Separate* process is implemented on top of FeatureHouse's composition operator in Java. We use the parsing process that was provided in FeatureHouse [4] to separate features to the FPH representation and added about 200 LOC.

The main process to check commutativity is implemented as a Python script in about 250 LOC. The first two parts of the commutativity check are directly implemented in the script. The third one, *Check shared variables*, requires considering possible aliases of feature-based Java programs. For this check, we have implemented a Java program, FPH_varsAnalysis, that calls Soot [21] to build the call graph and analyze each reachable method. FPH_varsAnalysis is an interprocedural context insensitive points-to analysis that, given two feature fragments that superimpose the same method, checks whether a variable of the same type is written by at least one of them and read or written by the other. Since feature fragments cannot be compiled by themselves (and thus Soot cannot be used on them), in order to do alias analysis, our program requires a representation that consists of the base system and all possible features. This representation is readily available for systems from the SPLVerifier repository since it uses a family-based approach to analysis. We generate a similar representation for all other systems used in our experiments.

Models and Methods. We have applied FPH to 29 case studies written in Java. In the first five columns of Table 1, we summarize the information about these systems. The first six have been considered by SPLVerifier [6] – a tool for checking whether a software product line (SPL) satisfies its feature specifications. SPLVerifier includes sample-based, product-based and family-based analyses and assumes that the order in which features should be composed is provided. The SPLVerifier examples came with specifications given by aspects woven at base system points, with an exception thrown if the state violates an expected property. The rest of our case studies are SPLs from the FeatureHouse repository [4].

We were unable to identify other techniques for analyzing feature commutativity of Java programs. Plath and Ryan [25] and Atlee et al. [8] compare

different composition orders but handle only state machines. SPLVerifier [6] represents state of the art in verification of feature-based systems expressed in Java, but it is not designed to do non-commutativity analysis. In the absence of alternative tools, we adapted SPLVerifier to the task of finding non-commutativity violations to be able to compare with FPH.

We conducted two experiments to evaluate FPH and to answer our research questions. For the first, we ran SPLVerifier on the first six systems (all properties that came with them satisfied the pattern in Sect. 2 and thus were appropriate for commutativity detection) presented in Table 1 to identify non-commutativity interactions. Since SPLVerifier is designed to check products against a set of specifications, we have to define what a commutativity check means in this context. For a pair of features, SPLVerifier would detect a commutativity violation if, upon composing these features in different orders, the provided property produces different values. During this check, SPLVerifier considers composition of all other features of the system in all possible orders and thus can identify two-way, three-way, etc. feature interactions, if applicable. We measured the time taken by SPLVerifier and the number of interactions found.

For the second experiment, we checked all 29 systems using FPH to identify non-commutativity interactions. We measured the number of feature pairs that required checking for shared variables, the time the analysis took and the precision of FPH in finding interactions. We were unable to establish ground truth for non-commutativity analysis in cases where FPH required the shared variables check due to our tool's reliance on Soot's unsound call graph construction [7]. Thus, we measured precision of our analysis by manually analyzing the validity of every interaction found by FPH. We also calculated SPLVerifier's *relative* recall, i.e., the ratio of non-commutativity-related interactions detected by FPH that were also detected by SPLVerifier. We did not encounter any interactions that were detected by SPLVerifier but not by FPH.

When the shared variables check is not necessary, our technique is sound. In such cases, if we inform the user that two features are commutative, they certainly are, and there is no need to define an order between them. As shown below, soundness was affected only for a small number of feature pairs. Moreover, advances in static analysis techniques may improve our results for those cases in the future. Our experiments were performed on a 2 GB RAM Virtual machine within an Intel Core i5 machine dual-core at 1.3 GHz.

Results. Columns 6–10 of Table 1 summarize results of our experiments, including, for the first six examples, SPLVerifier's precision and (relative) recall. "SV pairs" capture the number of feature pairs for which the shared variables check was required. A dash in the precision columns means that the measurement was not meaningful since no interactions were detected. E.g., SPLVerifier does not detect any non-commutativity interactions for Email, and FPH does not find any non-commutativity interactions for EPL. FPH found a number of instances of non-commutativity such as the one between *ExecutiveFloor* and *TwoThirdsFull* in the Elevator System. Only one SV check was required (while checking *Empty* and *Weight* features). Without our technique, the user would need to

Table 1. Overview of case studies.

System	LOC	# Feat.	# Frag.	Description	# Comm Interactions	SV Pairs	FPH Precision	SPLV Precision	SPLV Rel. Recall
Elevator	799	5	19	Our running example	1	1	1	1	1
Email	938	8	55	Email communication suite	3	9	1	-	0
Minepump	425	6	10	Water pump in mining operation	3	0	1	1	0.67
GPL	2510	17	109	Graph product line	2	38	0.1	-	0
AJStats	15311	19	128	Statistics for AspectJ	26	136	1	-	0
ZipMe	5479	12	229	Zip compression library	5	0	1	-	0
BerkeleyDB	64652	98	2667	Embedded database engine	198	1	1		
ChatSystem/Burke	614	7	51	Network client and server	2	14	0.33		
ChatSystem/Dreiling	938	5	78	Network client and server	3	0	1		
ChatSystem/Becker	651	6	42	Network client and server	5	2	1		
ChatSystem/Weiss	931	9	23	Network client and server	4	5	0.75		
ChatSystem/Schink	873	6	50	Network client and server	4	1	1		
ChatSystem/Rehn	862	6	58	Network client and server	14	2	1		
ChatSystem/Thuem	544	7	34	Network client and server	1	2	1		
EPL	99	10	22	Arithmetic expression evaluator	0	1	-		
GameOfLife	1656	14	154	Computer game	5	0	1		
Graph	467	4	26	Graph library	0	6	-		
Notepad/Quark	1397	11	106	Text editor	20	21	1		
Notepad/Delaware	1654	5	122	Text editor	10	0	1		
Notepad/Wellington	1522	3	38	Text editor	0	0	-		
Notepad/Svetoslav	1627	5	83	Text editor	0	0	-		
Notepad/Wehrman	1716	4	83	Text editor	6	6	1		
Notepad/Guimbarda	1586	14	229	Text editor	91	0	1		
Notepad/Robison	1404	9	90	Text editor	0	0	-		
PKJab	4994	7	99	Chat network client	2	0	1		
Raroscope	428	4	18	Compression library	0	0	-		
Sudoku	1850	6	103	Computer game	5	4	1		
TankWar	3184	19	213	Computer game	71	27	0.97		
Violet	9789	87	912	UML model editor	35	28	1		

provide order between the five features of the Elevator System, that is, specify 20 (5×4) ordering constraints. FPH allows us to conclude that *ExecutiveFloor* and *TwoThirdsFull* do not commute, that *Empty* and *Weight* likely commute but this is not guaranteed, and that all other pairs of features do commute. Thus, only two feature pairs required further analysis by the user.

The Minepump system did not require the shared variable check at all and thus FPH analysis for it is sound, and all three of the found interactions were manually confirmed to be "real" (thus, precision is 1). ChatSystem/Weiss has nine features which would imply needing to define the order between 72 (9×8) feature pairs. Four non-commutativity cases were found, all using the shared variables check, but only three were confirmed as "real" via a manual inspection (thus, precision is 0.75). We conclude that FPH is effective in discovering non-commutativity violations and proving their absence (**RQ1**).

We now turn to studying the accuracy of FPH w.r.t. finding non-commutativity violations (**RQ2**). From Table 1, we observe that for the Elevator System, both FPH and SPLVerifier correctly detect a non-commutativity interaction. For the Minepump system, SPLVerifier only finds two out of the three interactions found by FPH (relative recall = 0.67). For the Email system, AJStats, ZipMe, and GPL the specifications available in SPLVerifier do not allow detecting any of the non-commutativity interactions found by FPH (relative recall = 0).

GPL was a problematic case for FPH, affecting its precision. The graph algorithms in this example take a set of vertices and create and maintain an internal

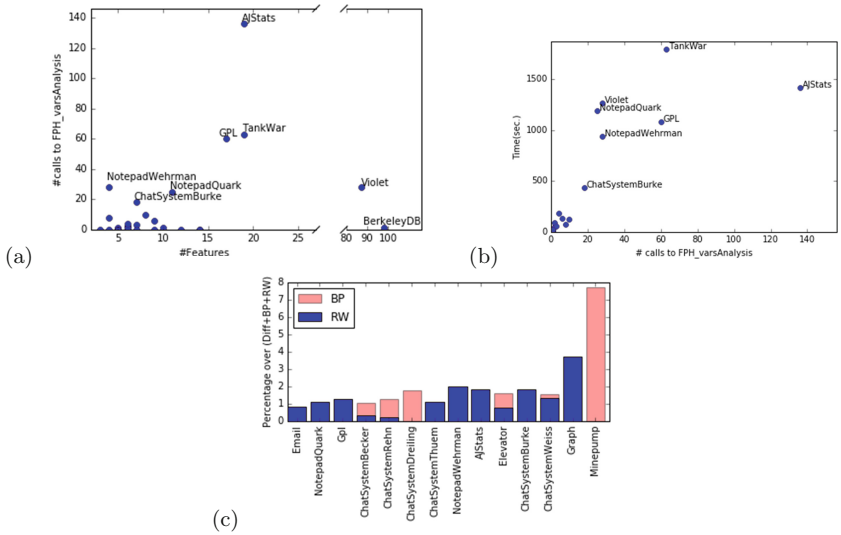


Fig. 6. (a) Number of FPH_varsAnalysis calls per system; (b) Time spent by FPH_varsAnalysis per system; (c) Percentage of non-commutativity checks where BP or SV analyses were applied last. (Color figure online)

data structure (e.g., to calculate the vertices involved in the shortest path or in a strongly connected component). With this data structure, our analysis found a number of possible shared variables and incorrectly deemed several features as non-commutative. E.g., the algorithms to find cycles or the shortest path between two nodes access the same set of vertices but change different fields and thus are commutative. One way of avoiding such false positives would be to implement field-sensitive alias analysis. While more precise, it will be significantly slower than our current shared variables analysis.

For the remaining systems, either FPH's reported interactions were "real", or, in cases where it returned some false positives (ChatSystemBurke, ChatSystemWeiss, and TankWar), it had to do with the precision of the alias analysis. Thus, given SPLVerifier's set of properties, FPH always exhibited the same or better precision and recall than SPLVerifier. Moreover, for all but three of the remaining systems, FPH exhibited perfect precision. We thus conclude that FPH is very accurate (**RQ2**).

We now turn to the efficiency of our analysis (**RQ3**). The time it took to separate features into behavior and composition was usually under 5 s. The outlier was BerkeleyDB, which took about a minute, due to the number of features and especially fragments (BerkeleyDB has 2667 fragments whereas Violet has 912 and the other systems have at most 229). In general, the time taken by FPH's commutativity check was highly influenced by the number of calls to FPH_varsAnalysis. Figure 6a shows the number of calls to FPH_varsAnalysis as the number of features increases. E.g., BerkeleyDB has 98 features and required

only one call to FPH_varsAnalysis, while AJStats has 19 features and required 136 of these calls. More features does not necessarily imply needing more of these checks. E.g., Violet and BerkeleyDB required fewer checks than AJStats, TankWar, and GPL, and yet they have more features.

Figure 6b shows the overall time spent by FPH_varAnalysis per system being analyzed. NotepadQuark and Violet took more time (resp., 1192 sec. and 1270 sec.) than GPL (1084 sec.) since these systems have calls to Java GUI libraries (awt and swing), thus resulting in a larger call graph than for GPL. A similar situation occurred during checking TankWar (1790 sec.) and AJStats (1418 sec.). It took FPH under 200 s in most cases and less than 35 min in the worst case to analyze non-commutativity (see Fig. 6b). FPH was efficient because FPH_varAnalysis was required for a relatively small fraction of pairs of feature fragments. We plot this information in Fig. 6c. For each analyzed system, it shows the percentage of feature fragments for which *behavior preservation* (BP) or *shared variables* (SV) was the last check conducted by FPH (out of the possible 100%). We omit the systems for which these checks were required for less than 1% of feature pairs. The figure shows that the calls to FPH_varsAnalysis (to compute SV, in blue) were not required for over 96% of feature pairs.

To check for non-commutativity violations, SPLVerifier needs to check all possible products which is infeasible in practice. So we set the timeout to one hour during which SPLVerifier was able to check 110 products for Elevator, 57 for Email, 151 for Minepump, 3542 for GPL, 2278 for AJStats and 1269 for ZipMe. For each of these systems, a different check is required for every specification, thus the same product is checked more than once if more than one specification exists. Even though GPL, AJStats and ZipMe are larger systems with more features, they have fewer properties associated with them and therefore we were able to check more products within one hour. Thus, to answer **RQ3**, FPH was much more efficient than SPLVerifier in performing non-commutativity analysis. SPLVerifier was only able to analyze products containing the base system and at most three features before reaching a timeout. Moreover, FPH can *guarantee commutativity*, while SPLVerifier cannot because of it being based on the properties given.

Our experiments also allow us to conclude that our technique is highly scalable (**RQ4**). E.g., the percentage of calls to FPH_varsAnalysis is shown to be small and increases only slightly with increase in the number of fragments (see Fig. 6a and b).

Threats to Validity. Our results may not generalize to other feature-based systems expressed in Java. We believe we have mitigated this threat by running our tool on examples provided by FeatureHouse. They include a variety of systems of different sizes which we consider to be representative of typical Java feature-based systems. As mentioned earlier, our use of SPLVerifier was not as intended by its designers. We also had no ground truth when the shared variable check was required. For those few cases, we calculated SPLVerifier's relative instead of actual recall.

5 Related Work

In this section, we survey related work on modular feature definitions, feature interaction detection and commutativity-related feature interactions.

Modular Feature Definitions. A number of approaches to modular feature definitions have been proposed. E.g., the composition language in [8] includes states in which the feature is to be composed (similar to our *fg.location*) and the feature behavior (similar to our *fb.body*). Other work [4,9,10] uses superimposition of FSTs to obtain the composed system. In [14,25], new variables are added or existing ones are changed with particular kind of compositions (either executing a new behavior when a particular variable is read, or adding a check before a particular variable is set). These approaches treat the feature behavior together with its composition specification. Instead, our approach automatically separates feature definition into the behavioral and the composition part, enabling a more scalable and efficient analysis.

Feature Interaction Detection. Calder et al. [13] survey approaches for analyzing feature interactions. Interactions occur because the behavior of one feature is being affected by others, e.g., by adding non-deterministic choices that result in conflicting states, by adding infinite loops that affect termination, or by affecting some assertions that are satisfied by the feature on its own. Checking these properties as well as those discussed in more recent work [8,15,18,19] requires building the entire SPL. Additionally, all these approaches consider state machine representations which are not available for most SPLs, and extracting them from code is non-trivial. SPLLift [12] is a family-based static analysis tool not directly intended to find interactions. Any change in a feature would require building the family-based representation again, whereas we conduct modular checks between features. Spek [26] is a product-based approach that analyzes whether the different products satisfy provided feature specifications. It does not check whether the features commute.

Non-commutativity-Related Feature Interactions. [5,8] also looked at detecting non-commutativity-related feature interactions. [5] presents a feature algebra and shows why composition (by superimposition) is, in general, not commutative. [8] analyzes feature commutativity by checking for bisimulation, and the result of the composition is a state machine representing the product. Neither work reports on a tool or applies to systems expressed in Java.

Aspect-Oriented Approaches. Storzer et al. [27] present a tool prototype for detecting precedence-related interactions in AspectJ. Technically, this approach is very similar to ours: it (a) detects which advice is activated at the same place; (b) checks whether the `proceed` keyword and exceptions are present; and (c) analyzes read and written variables. Yet, the focus is on aspects, and often many aspects are required to implement a single feature [23]. This implies that for m features with an average of n aspects each, the analysis in [27] needs to make $\mathcal{O}\left((m \cdot n)^2\right)$ checks, while our approach requires $\mathcal{O}\left(m^2\right)$ checks. Therefore, the approach in [27] might be significantly slower than FPH. [1] analyzes

interactions of aspects given by composition filters by checking for simulation among all the different orderings in which advice with shared joinpoints can be composed. As the number of advice with shared joinpoints increases, that approach considers every possible ordering, while we keep the analysis pairwise. [16,20] define modular techniques to check properties of aspect-oriented systems. [16] uses assume-guarantee reasoning to verify and detect interactions even when aspects can be activated within other aspects. It does not require an order but does require specifications to detect whether a certain composition order would not satisfy these. [20] uses the explicit CTL model-checking algorithm to distribute global properties into local properties to be checked for each aspect. This yields a modular check. In addition to requiring specifications, this technique assumes AspectJ's ordering of aspects.

6 Conclusion and Future Work

In this paper, we presented a compositional approach for checking non-commutativity of features in systems expressed in Java. The method is based on determining whether pairs of features can write to the same variables and thus the order in which features are composed to the base system may determine their valuation. The method is complementary to other feature interaction detection approaches such as [6,12] in that it helps build an order in which features are to be composed. When two features commute, they can be composed in any order. In addition, this method helps detect a number of feature interactions. The method is implemented in our framework FPH – Mr. Feature Potato Head. FPH does not require specifying properties of features and does not need to consider the entire set of software products every time a feature is modified. By performing an extensive empirical evaluation of FPH, we show that the approach is highly scalable and effective. In the future, we plan to further evaluate our technique, handle languages outside of Java and experiment with more precise methods for determining shared variables.

Acknowledgements. We thank anonymous reviewers for their helpful comments. This research has been supported by NSERC.

References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proceedings of AOSD 2009, pp. 39–50 (2009)
2. Apel, S., Atlee, J., Baresi, L., Zave, P.: Feature interactions: the next generation (Dagstuhl Seminar 14281). Dagstuhl Rep. **4**(7), 1–24 (2014)
3. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)
4. Apel, S., Kastner, C., Lengauer, C.: FeatureHouse: language-independent, automated software composition. In: Proceedings of ICSE 2009, pp. 221–231 (2009)

5. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebra for features and feature composition. In: Proceedings of AMAST 2008, pp. 36–50 (2008)
6. Apel, S., Von Rhein, A., Wendler, P., Groslinger, A., Beyer, D.: Strategies for product-line verification: case studies and experiments. In: Proceedings of ICSE 2013 (2013)
7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Not. **49**(6), 259–269 (2014)
8. Atlee, J., Beidu, S., Fahrenberg, U., Legay, A.: Merging features in featured transition systems. In: Proceedings of MoDeVVa@MODELS 2015, pp. 38–43 (2015)
9. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE TSE **30**(6), 355–371 (2004)
10. Beidu, S., Atlee, J., Shaker, P.: Incremental and commutative composition of state-machine models of features. In: Proceedings of MiSE@ICSE 2015, pp. 13–18 (2015)
11. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature?: A qualitative study of features in industrial software product lines. In: Proceedings of SPLC 2015, pp. 16–25 (2015)
12. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: SPLLift: Statically analyzing software product lines in minutes instead of years. In: Proceedings of PLDI 2013, pp. 355–364 (2013)
13. Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: Feature interaction: A critical review and considered forecast. Comput. Netw. **41**(1), 115–141 (2003)
14. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Sci. Comput. Program. **80**, 416–439 (2014)
15. Cordy, M., Classen, A., Schobbens, P.-Y., Heymans, P., Legay, A.: Managing evolution in software product lines: a model-checking perspective. In: Proceedings of VaMoS 2002, pp. 183–191 (2012)
16. Disenfeld, C., Katz, S.: A closer look at aspect interference and cooperation. In: Proceedings of AOSD 2012, pp. 107–118. ACM (2012)
17. Fantechi, A., Gnesi, S., Semini, L.: Optimizing feature interaction detection. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) FMICS-AVoCS 2017. LNCS, vol. 10471, pp. 201–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_13
18. Guelev, D., Ryan, M., Schobbens, P.-Y.: Model-checking the preservation of temporal properties upon feature integration. STTT **9**(1), 53–62 (2007)
19. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_11
20. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: ACM SIGSOFT SEN, vol. 29, pp. 137–146. ACM (2004)
21. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Proceedings of CETUS 2011, vol. 15, p. 35 (2011)
22. Liu, J., Batory, D., Nedunuri, S.: Modeling interactions in feature oriented software designs. In: Proceedings of ICFI 2005 (2005)

23. Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating support for features in advanced modularization technologies. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 169–194. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_8
24. Nipkow, T., Von Oheimb, D.: Java$_{light}$ is type-safe - definitely. In: Proceedings of PLDI 1998, pp. 161–170. ACM (1998)
25. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
26. Scholz, W., Thüm, T., Apel, S., Lengauer, C.: Automatic detection of feature interactions using the java modeling language: an experience report. In: Proceedings of SPLC 2011, p. 7 (2011)
27. Storzer, M., Forster, F.: Detecting precedence-related advice interference. In: Proceedings of ASE 2006, pp. 317–322, September 2006
28. Zave, P.: Feature interactions and formal specifications in telecommunications. IEEE Comput. **26**(8), 20–29 (1993)